**重庆大学课程设计报告**

课程设计题目： MIPS SOC 设计与性能优化

学 院： 计算机学院

专业班级： 计科6班 信安1班

年 级： 2019

姓 名： 王谦铭 王渝森

学 号： 20195456 20194199

完成时间： 2022 年 1 月 8 日

成 绩：

指导教师： 吴长泽

重庆大学教务处制

综合设计指导教师评定成绩表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 项目 | 分值 | 优秀  (100>x≥90) | 良好  (90>x≥80) | 中等  (80>x≥70) | 及格  (70>x≥60) | 不及格(x<60) | 评分 |
| 参考标准 | 参考标准 | 参考标准 | 参考标准 | 参考标准 |
| 学习态度 | 15 | 学习态度认真，科学作风严谨，严格保证设计时间并按任务书中规定的进度开展各项工作 | 学习态度比较认真，科学作风良好，能按期圆满完成任务书规定的任务 | 学习态度尚好，遵守组织纪律，基本保证设计时间，按期完成各项工作 | 学习态度尚可，能遵守组织纪律，能按期完成任务 | 学习马虎，纪律涣散，工作作风不严谨,不能保证设计时间和进度 |  |
| 技术水平与实际能力 | 25 | 设计合理、理论分析与计算正确，实验数据准确，有很强的实际动手能力、经济分析能力和计算机应用能力，文献查阅能力强、引用合理、调查调研非常合理、可信 | 设计合理、理论分析与计算正确，实验数据比较准确，有较强的实际动手能力、经济分析能力和计算机应用能力，文献引用、调查调研比较合理、可信 | 设计合理，理论分析与计算基本正确，实验数据比较准确，有一定的实际动手能力，主要文献引用、调查调研比较可信 | 设计基本合理，理论分析与计算无大错，实验数据无大错 | 设计不合理，理论分析与计算有原则错误，实验数据不可靠，实际动手能力差，文献引用、调查调研有较大的问题 |  |
| 创新 | 10 | 有重大改进或独特见解，有一定实用价值 | 有较大改进或新颖的见解，实用性尚可 | 有一定改进或新的见解 | 有一定见解 | 观念陈旧 |  |
| 论文(计算书、图纸)撰写质量 | 50 | 结构严谨，逻辑性强，层次清晰，语言准确，文字流畅，完全符合规范化要求，书写工整或用计算机打印成文；图纸非常工整、清晰 | 结构合理，符合逻辑，文章层次分明，语言准确，文字流畅，符合规范化要求，书写工整或用计算机打印成文；图纸工整、清晰 | 结构合理，层次较为分明，文理通顺，基本达到规范化要求，书写比较工整；图纸比较工整、清晰 | 结构基本合理，逻辑基本清楚，文字尚通顺，勉强达到规范化要求；图纸比较工整 | 内容空泛，结构混乱，文字表达不清，错别字较多，达不到规范化要求；图纸不工整或不清晰 |  |

指导教师评定成绩：

指导教师签名： 年 月 日

**MIPS SOC设计报告**

王谦铭、王渝森

一、设计简介

设计内容：MIPS SOC设计与性能优化。

实现的功能：MIPS 52 条基础指令，包括14条算术运算指令、8条逻辑运算指令、6条移位指令、12条分支指令、4条数据移动指令、8条访存指令。并完善冒险模块，设计异常处理模块。

实现方式：基于计组实验4实现的简单流水线CPU，完成对52条指令的添加实现。

实现效果：52条指令独立测试全部通过

1. 小组分工说明

王渝森：负责逻辑运算指令，部分算术运算指令，代码调试和纠错

王谦铭：负责其余所有指令，代码调试和纠错

（二）设计任务的问题分析

对于我们小组来说，面对的任务有两个，添加到52条指令，以及封装。

首先，对于52条指令，在上学期的实验中我们已经学到了很多，并且本学期深入学习了流水线处理的问题。52条指令只需要分析指令所需要的功能，以及判断指令的格式，并且告诉对应的模块怎么处理指令，还有注意流水，即可。另外还要注意一些新增指令导致的冲突。

封装上，我们看了ppt的教程内容，分析了封装需要增加的数据连线，以及debug的方式，明白这些内容，封装问题就迎刃而解了。

二、设计方案（30%）

（一）总体设计思路

总体设计思路：从系统顶层角度出发，CPU包括mips模块。mips中的数据通路datapath模块和控制器controller模块完成对指令的接收，对指令的译码，输出控制信号，并完成对数据的处理、运算、指令跳转等功能，datapath模块中又包含了冒险模块等来完成相应的功能。

模块划分：

1.mycpu\_top模块：该顶层模块连接了mips模块。其中mips模块包括controller模块和datapath模块。

2.controller模块：该模块负责接收得到的指令并译码,输出控制信号，对于不同阶段输出的控制信号不同。主要的信号有控制分支的jump、pcsrc等，控制操作数选择的alusrc，控制写入的寄存器信号regdst，以及各类存储模块的写入信号regwriteW、hilowriteW等。

3.datapath模块：该模块连接了CPU的寄存器堆，冒险模块，HILO寄存器等等，是CPU的核心。

4.harzard模块：负责处理各类可能产生的冒险，在计组实验4的基础上，需要注意对hilo寄存器的数据冒险的解决。总的来说，需要实现解决如下冒险，如数据冒险（包括需要访问regfile、hilo寄存器数据时产生的数据冒险）、控制冒险（分支跳转指令产生的冒险）等等，输出相应的暂停信号、前推信号以及各级流水线对应的清空信号。

1. controller模块设计

功能意图：

controller模块负责指令的译码，产生相应的控制信号。

输入输出：

|  |  |  |  |
| --- | --- | --- | --- |
| 信号量 | 输入输出 | 位宽 | 功能描述 |
| clk | input | 1 | 时钟信号：0表示下降沿，1表示上升沿 |
| rst | input | 1 | 复位信号：0表示不复位，1表示复位 |
| opD | input | 6 | D阶段指令的[31:26]字段，指令的标识符之一 |
| functD | input | 6 | D阶段指令的[5:0]字段，指令的标识符之一 |
| pcsrcD | output | 1 | 用来控制下一个PC |
| branchD | output | 1 | 表示是否为分支跳转指令（D阶段信号） |
| equalD | input | 1 | 判断rs rt两个寄存器的值在D阶段是否相等，用来配合branchD |
| jumpD | output | 1 | 表示是否为跳转指令（D阶段信号） |
| flushE | input | 1 | 刷新D-E阶段的触发器 |
| memtoregE | output | 1 | 控制写回的值，是来自内存，还是来自alu的计算结果 |
| alusrcE | output | 1 | 控制ALU的第二个输入是立即数还是寄存器里的数（或即将写回寄存器的数） |
| regdstE | output | 1 | 写回的目的寄存器来自rt位还是rd位 |
| regwriteE | output | 1 | 表示是否要写寄存器堆信号（E阶段信号） |
| alucontrolE | output | 8 | 控制alu进行什么运算的信号 |
| memtoregM | output | 1 | 控制写回的值，是来自内存，还是来自alu的计算结果（M阶段） |
| memwriteM | output | 1 | 控制写不写如内存 |
| regwriteM | output | 1 | 表示是否要写寄存器堆信号（M阶段信号） |
| memtoregW | output | 1 | 控制写回的值，是来自内存，还是来自alu的计算结果（W阶段） |
| regwriteW | output | 1 | 表示是否要写寄存器堆信号（W阶段信号） |
| hiwriteE | output | 1 | hi寄存器的写信号（E阶段信号） |
| lowriteE | output | 1 | lo寄存器的写信号（E阶段信号） |
| hiwriteM | output | 1 | hi寄存器的写信号（M阶段信号） |
| lowriteM | output | 1 | lo寄存器的写信号（M阶段信号） |
| hireadE | output | 1 | hi寄存器的读信号（E阶段信号） |
| loreadE | output | 1 | lo寄存器的写信号（E阶段信号） |
| multE | output | 1 | 没用上，设置以防万一的 |
| multM | output | 1 | M阶段传给hilo寄存器的，控制是否写入 |
| stallE | input | 1 | 暂停E-M阶段触发器 |
| rsGreateror  EqualZeroD | input | 1 | 判断rs寄存器是不是大于等于0 |
| rtD | input | 5 | 指令[20:16]字段，寄存器号 |
| rsGreaterZeroD | input | 1 | 判断rs寄存器是不是大于0 |
| rsLessor  EqualZeroD | input | 1 | 判断rs寄存器是不是小于等于0 |
| rsLessZeroD | input | 1 | 判断rs寄存器是不是小于0 |
| jrD | output | 1 | 如果是jr，那么pc就要设置成rs寄存器里的值，下次跳转跳到那里 |
| jalD | output | 1 | 没用上 |
| jalE | output | 1 | 控制alu的两端输入，如果是jal，那么a端口输入pc值，b端口输入立即数8。同时也控制着写回的寄存器号，如果是jal，就写入31号寄存器 |
| jalrD | output | 1 | 没用上 |
| jalrE | output | 1 | 控制alu的两端输入，如果是jalr，那么a端口输入pc值，b端口输入立即数8。 |

实现逻辑：

产生的控制信号主要包括了两个部分：maindec产生的各类控制信号，包括分支跳转控制、目的寄存器选择控制、源操作数选择控制、各类写使能信号；另一种是alu\_dec产生控制alu操作的alucontrol信号。

因为alu模块主要描述了对于指令中的不同op,funct或者rs,rt的值，给alucontrol不同的信号。其中部分指令仅从op即可判断出指令要进行的操作，但部分指令是从op和funct共同判断的，因此使用的是case(op),接所有与op相关的指令，default:case(funct)后再接与funct相关的指令。

而对于较为特殊的指令，如比较跳转等需要对寄存器rt中的数值先进性判断。指令控制信号alucontrol的二进制编码使用提供的宏定义文件defines.vh中，直接调用即可。

核心代码：

module controller(

    input wire clk,rst,

    //decode stage

    input wire[5:0] opD,functD,

    output wire pcsrcD,branchD,

    input wire equalD,

    output wire jumpD,

    //execute stage

    input wire flushE,

    output wire memtoregE,alusrcE,

    output wire regdstE,regwriteE,

    output wire[7:0] alucontrolE,

    //mem stage

    output wire memtoregM,memwriteM,

                regwriteM,

    //write back stage

    output wire memtoregW,regwriteW,

    output wire hiwriteE,lowriteE,hiwriteM,lowriteM,

    output wire hireadE,loreadE,

    output wire multE,multM,

    input wire stallE,

    input wire rsGreaterOrEqualZeroD,

    input wire [4:0] rtD,

    input wire rsGreaterZeroD,

    input wire rsLessOrEqualZeroD,

    input wire rsLessZeroD,

    output wire jr,//传递给数据通路的jr信号，因为跳转相关都在数据通路中执行

    output wire jalD,jalE,//传递给数据通路

    output wire jalrD,jalrE,

    output wire lbW,lbuW,lhW,lhuW

    );

    //decode stage

    wire memtoregD,memwriteD,alusrcD,

        regdstD,regwriteD;

    wire[7:0] alucontrolD;

    //execute stage

    wire memwriteE;

    wire hiwriteD,lowriteD;

    wire hireadD,loreadD;

    wire beq;//beq信号

    wire bne;//bne信号

    wire bgez;//bgez信号

    wire bgtz;//bgtz信号

    wire blez;//blez信号

    wire bltz;//bltz信号

    wire lbD,lbE,lbM,lbW,lbuD,lbuE,lbuM,lbuW;

    wire lhD,lhE,lhM,lhW,lhuD,lhuE,lhuM,lhuW;

    maindec md(

        opD,functD,

        memtoregD,memwriteD,

        branchD,alusrcD,

        regdstD,regwriteD,

        jumpD,

        hiwriteD,lowriteD,

        hireadD,loreadD,

        multD,

        beq,

        bne,

        bgez,

        rtD,

        bgtz,

        blez,

        bltz,

        jr,

        jalD,

        jalrD,

        lbD,lbuD

        );

    aludec ad(functD,opD,alucontrolD,jalD);

    //beq信号，并且两个数相等               bne信号，并且两个数不等   bgez信号，并且rs大于等于0           bgtz信号，并且rs大于0                  blez信号，并且rs小于等于0             bltz信号，并且rs小于0

    assign pcsrcD = (branchD &beq & equalD)|(branchD & bne & ~equalD)|(branchD & bgez & rsGreaterOrEqualZeroD)|(branchD & bgtz & rsGreaterZeroD)|(branchD & blez & rsLessOrEqualZeroD)|(branchD & bltz & rsLessZeroD);

    //pipeline registers

    flopenrc #(24) regE(

        clk,

        rst,

        ~stallE,

        flushE,

        {memtoregD,memwriteD,alusrcD,regdstD,regwriteD,alucontrolD,hiwriteD,lowriteD,hireadD,loreadD,multD,jalD,jalrD,lbD,lbuD,lhD,lhuD},

        {memtoregE,memwriteE,alusrcE,regdstE,regwriteE,alucontrolE,hiwriteE,lowriteE,hireadE,loreadE,multE,jalE,jalrE,lbE,lbuE,lbE,lbuE}

        );

    flopr #(16) regM(

        clk,rst,

        {memtoregE,memwriteE,regwriteE,hiwriteE,lowriteE,multE,lbE,lbuE,lhE,lhuE},

        {memtoregM,memwriteM,regwriteM,hiwriteM,lowriteM,multM,lbM,lbuM,lhM,lhuM}

        );

    flopr #(12) regW(

        clk,rst,

        {memtoregM,regwriteM,lbM,lbuM,lhM,lhuM},

        {memtoregW,regwriteW,lbW,lbuW,lbM,lbuM}

        );

endmodule

1. datapath模块设计

功能意图：

连接CPU的寄存器堆，冒险模块， HILO寄存器等等，完成MIPS五级流水线取指、译码、执行、访存、回写五个阶段，是整个CPU的关键部分。

输入输出：

|  |  |  |  |
| --- | --- | --- | --- |
| 信号量 | 输入输出 | 位宽 | 功能 |
| clk | input | 1 | 时钟信号：0表示下降沿，1表示上升沿 |
| rst | input | 1 | 复位信号：0表示不复位，1表示复位 |
| pcF | output | 32 | 程序计数器，用于保存指令地址（F阶段信号） |
| instrF | input | 32 | 指令的二进制表示（F阶段信号） |
| pcsrcD | input | 1 | 同controller表 |
| branchD | input | 1 | 表示是否为分支跳转指令（D阶段信号） |
| jumpD | input | 1 | 表示是否为跳转指令（D阶段信号） |
| equalD | output | 1 | 同controller表 |
| opD | output | 6 | D阶段指令的[31:26]字段，指令的标识符之一 |
| functD | output | 6 | D阶段指令的[5:0]字段，指令的标识符之一 |
| mentoregE | input | 1 | 同controller表 |
| alusrcE | input | 1 | 同controller表 |
| regdstE | input | 1 | 同controller表 |
| regwriteE | input | 1 | 同controller表 |
| alucontrolE | input | 8 | 同controller表 |
| flushE | output | 1 | 同controller表 |
| memtoregM | input | 1 | 同controller表 |
| regwriteM | input | 1 | 同controller表 |
| aluoutM | output | 32 | alu的运算结果（M阶段信号） |
| writedata2M | output | 32 | 经过处理的writedata，原来直接写入内存即可，增加指令后需要进行扩展 |
| readdataM | input | 32 | 同controller表 |
| memtoregW | input | 1 | 同controller表 |
| regwriteW | input | 1 | 同controller表 |
| hiwriteE | input | 1 | hi寄存器的写信号（E阶段信号） |
| lowriteE | input | 1 | lo寄存器的写信号（E阶段信号） |
| hiwriteM | input | 1 | hi寄存器的写信号（M阶段信号） |
| lowriteM | input | 1 | lo寄存器的写信号（M阶段信号） |
| hireadE | input | 1 | hi寄存器的读信号（E阶段信号） |
| loreadE | input | 1 | lo寄存器的写信号（E阶段信号） |
| multE | input | 1 | 同controller表 |
| multM | input | 1 | 同controller表 |
| stallE | output | 1 | 暂停E-M阶段触发器 |
| rsGreaterorEqual  ZeroD | output | 1 | 同controller表 |
| rtD | output | 5 | 指令[20:16]字段，寄存器号 |
| rsGreaterZeroD | output | 1 | 同controller表 |
| rsLessorEqualZeroD | output | 1 | 同controller表 |
| rsLessZeroD | output | 1 | 同controller表 |
| jrD | input | 1 | 同controller表 |
| jalD | input | 1 | 同controller表 |
| jalE | input | 1 | 同controller表 |
| jalrD | input | 1 | 同controller表 |
| jalrE | input | 1 | 同controller表 |
| mensel | output | 4 | 控制内存是写哪个位置 |
| memce | output | 1 | 恒为1，没啥用，因为只实现了52条指令 |

实现逻辑：

根据MIPS五级流水线，通路分为五个阶段：取指F、译码D、运算E、访存MEM、回写WB。

取指F阶段:将当前PC输出到指令存储器，并得到读取的指令。之后PC更新，更新的情况根据指令会有不同：

当是普通的运算类指令，不涉及指令的分支跳转时，新的PC加4即可。

当遇到分支跳转指令时，增加3个二选一的多选器判断PC的跳转，其中pcbrmux代表是否选择的是普通分支类指令，pcmux1代表是否选择普通的jump类指令，pcmux2判断是否选择跳转寄存器内地址。

译码ID阶段：首先触发器将上个阶段所需的信号和数据传递到此阶段。同时将指令分成更小的部分，如操作码、寄存器号、立即数等等,将各分好的部分传递给controller模块进行译码。

另外，立即数的符号/无符号扩展、移位操作，寄存器数据读取（regfile、hilo）也在此进行。由于是否进行分支的判断在此进行，需要判断的操作数就可能会产生数据冒险，故在这里进行了数据的前推:操作数可能在上一条或上上条计算出，但尚未被写入，因此前推了运算（Execute）阶段和访存（Memory）阶段的数据，根据数据来源的不同，需要在alu运算结果、数据存储器读出结果。

ALU运算EX阶段：

首先触发器将上个阶段所需的信号和数据传递到此阶段。这个阶段主要负责进行操作数之间的运算。

ALU的操作数可能来源有：寄存器堆、hilo寄存器，因此需要多选器来控制选择哪一个操作数。

由于需要操作数进行运算，可能会产生数据冒险。因此在这个阶段设置了几条旁路，分别来自上一条和上上条指令的结果。

根据视频讲解中对于除法器的实现，由于除法是使用了一个单独模块(div.v)来进行运算，因此控制除法运算的状态机和除法模块放置在了alu的外部来单独执行。

访存MEM阶段：首先触发器将上个阶段所需的信号和数据传递到此阶段。该阶段主要处理的是访存的信号。

写回WB阶段：首先触发器会将上个阶段所需的信号和数据传递到此阶段。这里控制回写寄存器的数据是哪一个数据(alu运算结果、数据存储器读取结果、除法器计算结果)。

关键代码：

module datapath(

    input wire clk,rst,

    //fetch stage

    output wire[31:0] pcF,

    input wire[31:0] instrF,

    //decode stage

    input wire pcsrcD,branchD,

    input wire jumpD,

    output wire equalD,

    output wire[5:0] opD,functD,

    //execute stage

    input wire memtoregE,

    input wire alusrcE,regdstE,

    input wire regwriteE,

    input wire[7:0] alucontrolE,

    output wire flushE,

    //mem stage

    input wire memtoregM,

    input wire regwriteM,

    output wire[31:0] aluoutM,writedata2M,

    input wire[31:0] readdataM,

    //writeback stage

    input wire memtoregW,

    input wire regwriteW,

    input wire hiwriteE,lowriteE,hiwriteM,lowriteM,

    input wire hireadE,loreadE,

    input wire multE,multM,

    output wire stallE,

    output wire rsGreaterOrEqualZeroD,

    output wire[4:0] rtD,//用来告诉controller，让它判断branch指令是哪条

    output wire rsGreaterZeroD,

    output wire rsLessOrEqualZeroD,

    output wire rsLessZeroD,

    input wire jrD,

    input wire jalD,jalE,

    input wire jalrD,jalrE,

    input wire lbW,lbuW,lhW,lhuW,

    output wire[3:0] memsel,

    output wire memce

    );

    wire [31:0] writedataM;

    //fetch stage

    wire stallF;

    //FD

    wire [31:0] pcnextFD,pcnextbrFD,pcplus4F,pcbranchD;

    wire [31:0] pcnextFDbeforeJR;

    //decode stage

    wire [31:0] pcplus4D,instrD;

    wire forwardaD,forwardbD;

    wire [4:0] rsD,rdD;

    wire flushD,stallD;

    wire [31:0] signimmD,signimmshD;

    wire [31:0] srcaD,srca2D,srcbD,srcb2D;

    //execute stage

    wire [1:0] forwardaE,forwardbE;

    wire [4:0] rsE,rtE,rdE;

    wire [4:0] writeregbeforeE;

    wire [4:0] writeregE;

    wire [31:0] signimmE;

    wire [31:0] srcaE,srca2E,srca3E,srcbE,srcb2E,srcb3E,srcb4E,srcb5E,srcb6E;

    wire [31:0] aluoutE;

    wire stallE;//暂停指令，因为除法器的加入而被引入

    wire divstallE;//由除法器发出的流水线暂停指令，当这条指令为1时，要控制前面的器件以及E都stall

    //mem stage

    wire [4:0] writeregM;

    //writeback stage

    wire [4:0] writeregW;

    wire [31:0] aluoutW,readdataW,resultW,resultbeforeW;

    wire [4:0] saD,saE;//新增的sa信号。6条移位运算指令需要它

    wire [31:0] regtohiD,regtohiE,regtohiM,in1hi,in2hi;

    wire [31:0] regtoloD,regtoloE,regtoloM,in1lo,in2lo;

    wire [31:0] hiout,loout;//hi lo寄存器里的数

    wire [31:0] alutohiE,alutoloE;//alu算出来的高低两个部分的位

    wire [31:0] alutohiM,alutoloM;

    wire div\_clear;

    assign div\_clear = 0;

    wire [63:0] div\_resultE;//用来记录除法器输出结果，此结果应流水到M阶段，交给hilo寄存器

    wire [63:0] div\_resultM;

    wire divE,divM;//div信号，用来控制hi lo寄存器的写入，divE信号由div组件产生

    wire [31:0] pcD,pcE;

    wire [7:0] alucontrolM;

    wire [31:0] writedata2M,readdata2M;

    wire mem\_we;

    //hazard detection

    hazard h(

        //fetch stage

        stallF,

        //decode stage

        rsD,rtD,

        branchD,

        forwardaD,forwardbD,

        stallD,

        //execute stage

        rsE,rtE,

        writeregE,

        regwriteE,

        memtoregE,

        forwardaE,forwardbE,

        flushE,

        //mem stage

        writeregM,

        regwriteM,

        memtoregM,

        //write back stage

        writeregW,

        regwriteW,

        divstallE,

        stallE

        );

    //next PC logic (operates in fetch an decode)

    mux2 #(32) pcbrmux(pcplus4F,pcbranchD,pcsrcD,pcnextbrFD);

    mux2 #(32) pcmux1(pcnextbrFD,

        {pcplus4D[31:28],instrD[25:0],2'b00},

        jumpD,pcnextFDbeforeJR);

    mux2 #(32) pcmux2(pcnextFDbeforeJR,

        srca2D,

        jrD,pcnextFD);

    //regfile (operates in decode and writeback)

    regfile rf(clk,regwriteW,rsD,rtD,writeregW,resultW,srcaD,srcbD);

    //fetch stage logic

    pc #(32) pcreg(clk,rst,~stallF,pcnextFD,pcF);

    adder pcadd1(pcF,32'b100,pcplus4F);

    //decode stage

    flopenr #(32) r1D(clk,rst,~stallD,pcplus4F,pcplus4D);

    flopenrc #(32) r2D(clk,rst,~stallD,flushD,instrF,instrD);

    flopenrc #(32) r3D(clk,rst,~stallD,flushD,pcF,pcD);

    signext se(instrD[15:0],instrD[29:28],signimmD);

    sl2 immsh(signimmD,signimmshD);

    adder pcadd2(pcplus4D,signimmshD,pcbranchD);

    mux2 #(32) forwardamux(srcaD,aluoutM,forwardaD,srca2D);

    mux2 #(32) forwardbmux(srcbD,aluoutM,forwardbD,srcb2D);

    eqcmp comp1(srca2D,srcb2D,equalD);

    greateroreqcmp comp2(srca2D,32'b0,rsGreaterOrEqualZeroD);//如果rs大于等于，就给一个rsGreaterOrEqualZeroD信号

    greatercmp comp3(srca2D,32'b0,rsGreaterZeroD);//如果rs大于0，就给一个rsGreaterZeroD信号

    lessoreqcmp comp4(srca2D,32'b0,rsLessOrEqualZeroD);//如果rs小于等于0，就给一个rsLessOrEqualZeroD信号

    lesscmp comp5(srca2D,32'b0,rsLessZeroD);//如果rs小于0，就给一个rsLessZeroD信号

    assign opD = instrD[31:26];

    assign functD = instrD[5:0];

    assign rsD = instrD[25:21];

    assign rtD = instrD[20:16];

    assign rdD = instrD[15:11];

    assign saD = instrD[10:6];//新增的sa信号。6条移位运算指令需要它

    assign regtohiD = srcaD;//需要写入hi的数

    assign regtoloD = srcaD;//需要写入lo的数

    //execute stage

    flopenrc #(32) r1E(clk,rst,~stallE,flushE,srcaD,srcaE);

    flopenrc #(32) r2E(clk,rst,~stallE,flushE,srcbD,srcbE);

    flopenrc #(32) r3E(clk,rst,~stallE,flushE,signimmD,signimmE);

    flopenrc #(5) r4E(clk,rst,~stallE,flushE,rsD,rsE);

    flopenrc #(5) r5E(clk,rst,~stallE,flushE,rtD,rtE);

    flopenrc #(5) r6E(clk,rst,~stallE,flushE,rdD,rdE);

    flopenrc #(5) r7E(clk,rst,~stallE,flushE,saD,saE);//新增的sa信号。6条移位运算指令需要它

    flopenrc #(32) r8E(clk,rst,~stallE,flushE,regtohiD,regtohiE);//需要写入hi的数

    flopenrc #(32) r9E(clk,rst,~stallE,flushE,regtoloD,regtoloE);//需要写入lo的数

    flopenrc #(32) r10E(clk,rst,~stallE,flushE,pcD,pcE);

    mux3 #(32) forwardaemux(srcaE,resultW,aluoutM,forwardaE,srca2E);

    mux3 #(32) forwardbemux(srcbE,resultW,aluoutM,forwardbE,srcb2E);

    mux2 #(32) srcajal(srca2E,pcE,jalE|jalrE,srca3E);

    mux2 #(32) srcbmux(srcb2E,signimmE,alusrcE,srcb3E);

    mux2 #(32) srcbmuxforhi(srcb3E,hiout,hireadE,srcb4E);

    mux2 #(32) srcbmuxforlo(srcb4E,loout,loreadE,srcb5E);//为输入数进行选择，是不是需要hi或者lo

    mux2 #(32) srcbjal(srcb5E,32'b1000,jalE|jalrE,srcb6E);

    alu alu(srca3E,srcb6E,alucontrolE,saE,aluoutE,alutohiE,alutoloE);

    mux2 #(5) wrmux(rtE,rdE,regdstE,writeregbeforeE);

    mux2 #(5) wrmux2(writeregbeforeE,5'b11111,jalE,writeregE);

    divWrapper div(

    .clk(clk), .rst(rst),

    .clear(div\_clear),

    .a(srca3E),

    .b(srcb6E),

    .op(alucontrolE),

    .div\_result(div\_resultE),

    .divstall(divstallE),

    .divE(divE)

    );

    //在e阶段进行除法运算

    //mem stage

    flopr #(32) r1M(clk,rst,srcb2E,writedataM);

    flopr #(32) r2M(clk,rst,aluoutE,aluoutM);

    flopr #(5) r3M(clk,rst,writeregE,writeregM);

    flopr #(32) r4M(clk,rst,regtohiE,regtohiM);

    flopr #(32) r5M(clk,rst,regtoloE,regtoloM);

    flopr #(32) r6M(clk,rst,alutohiE,alutohiM);

    flopr #(32) r7M(clk,rst,alutoloE,alutoloM);

    flopr #(64) r8M(clk,rst,div\_resultE,div\_resultM);

    flopr #(1) r9M(clk,rst,divE,divM);//div信号的流水

    flopr #(8) r10M(clk,rst,alucontrolE,alucontrolM);

    mux2 #(32) muxfor1hi(regtohiM,alutohiM,multM,in1hi);

    mux2 #(32) muxfor1lo(regtoloM,alutoloM,multM,in1lo);//选择器，选择是乘法算出来的传给hi lo还是寄存器传给hi alucontrolE

    mux2 #(32) muxfor2hi(in1hi,div\_resultM[63:32],divM,in2hi);

    mux2 #(32) muxfor2lo(in1lo,div\_resultM[31:0],divM,in2lo);//选择器，选择是之前的还是除法器的结果

    hilo\_reg hilo\_reg(clk,rst,hiwriteM|divM|mulM,lowriteM|divM|multM,in2hi,in2lo,hiout,loout);

    changedatamem mem1M(alucontrolM,writedataM,aluoutM,readdataM,writedata2M,readdata2M,memsel,mem\_we,mem\_ce);

    //writeback stage

    flopr #(32) r1W(clk,rst,aluoutM,aluoutW);

    flopr #(32) r2W(clk,rst,readdata2M,readdataW);

    flopr #(5) r3W(clk,rst,writeregM,writeregW);

    mux2 #(32) resmux(aluoutW,readdataW,memtoregW,resultW);

endmodule

1. hazard模块设计

功能意向：冒险模块对于保证流水线的正确执行起着十分关键的作用，通过解决冒险，确保了读取和写入的数据是正确的。主要就是利用要操作的寄存器号和各个阶段的控制信号，判断是否需要产生前推信号、暂停信号和清除信号，通过这些信号来消除冒险。

输入输出：

|  |  |  |  |
| --- | --- | --- | --- |
| 信号量 | 输入输出 | 位宽 | 功能 |
| stallF | output | 1 | F阶段暂停 |
| rsD | input | 5 | Rs在D阶段的寄存器地址 |
| rtD | input | 5 | Rt在D阶段的寄存器地址 |
| branchD | input | 1 | 是否跳转 |
| forwardaD | output | 1 | 对rsD的数据前推信号 |
| forwardbD | output | 1 | 对rtD的数据前推信号 |
| stallD | output | 1 | D阶段暂停 |
| rsE | input | 5 | Rt在E阶段的寄存器地址 |
| rtE | input | 5 | Rt在E阶段的寄存器地址 |
| writeregE | input | 5 | E阶段写回的寄存器的地址 |
| regwriteE | input | 1 | 是否写回寄存器 |
| memtoregE | input | 1 | 是否从内存里读 |
| forwardaE | output | 2 | 为alu的第一个输入控制转发 |
| forwardbE | output | 2 | 为alu的第二个输入控制转发 |
| flushE | output | 1 | E阶段刷新信号 |
| writeregM | input | 5 | M阶段写回寄存器的地址 |
| regwriteM | input | 1 | 控制是否写回寄存器 |
| memtoregM | input | 1 | 控制是否从内存里读 |
| writeregW | input | 5 | W阶段写回的寄存器的地址 |
| regwriteW | input | 1 | 是否写回寄存器 |
| divstallE | input | 1 | 除法暂停 |
| stallE | output | 1 | E阶段暂停信号 |

逻辑实现：

前推信号：包括普通运算操作数的前推、分支类所需判断操作数或需要按寄存器地址跳转的前推、hilo寄存器数据前推。

普通运算的前推有两种情况：第一是上一条指令刚执行到访存阶段，结果还未写入寄存器；第二是上上条指令准备回写，但还未写入。因此根据现在执行指令的操作数来源的寄存器号和writeregM和regwriteM、writeregW和regwriteW信号为依据判断前推。

hilo寄存器数据的前推逻辑与普通运算指令差别不大。

暂停信号：设计中去掉了原本分支指令的暂停信号，改为不暂停实现的分支判断。而只有当前一条是lw类指令，后一条马上需要lw类指令取出的结果时，才需要暂停。因为lw指令在访存阶段才会读出结果，而后一条指令在EX阶段就需要用到lw读出的结果。

此外，由于除法指令需要多个周期，因此除法状态机会传入一个除法暂停信号div\_stall.所以最终暂停信号逻辑如下：

stallE = divstallE;

stallD = lwstallD | branchstallD | stallE;

stallF = stallD;

关键代码：

module hazard(

    //fetch stage

    output wire stallF,

    //decode stage

    input wire[4:0] rsD,rtD,

    input wire branchD,

    output wire forwardaD,forwardbD,

    output wire stallD,

    //execute stage

    input wire[4:0] rsE,rtE,

    input wire[4:0] writeregE,

    input wire regwriteE,

    input wire memtoregE,

    output reg[1:0] forwardaE,forwardbE,

    output wire flushE,

    //mem stage

    input wire[4:0] writeregM,

    input wire regwriteM,

    input wire memtoregM,

    //write back stage

    input wire[4:0] writeregW,

    input wire regwriteW,

    input wire divstallE,

    output wire stallE

    );

    wire lwstallD,branchstallD;

    //forwarding sources to D stage (branch equality)

    assign forwardaD = (rsD != 0 & rsD == writeregM & regwriteM);

    assign forwardbD = (rtD != 0 & rtD == writeregM & regwriteM);

    //forwarding sources to E stage (ALU)

    always @(\*) begin

        forwardaE = 2'b00;

        forwardbE = 2'b00;

        if(rsE != 0) begin

            /\* code \*/

            if(rsE == writeregM & regwriteM) begin

                /\* code \*/

                forwardaE = 2'b10;

            end else if(rsE == writeregW & regwriteW) begin

                /\* code \*/

                forwardaE = 2'b01;

            end

        end

        if(rtE != 0) begin

            /\* code \*/

            if(rtE == writeregM & regwriteM) begin

                /\* code \*/

                forwardbE = 2'b10;

            end else if(rtE == writeregW & regwriteW) begin

                /\* code \*/

                forwardbE = 2'b01;

            end

        end

    end

    //stalls

    assign #1 lwstallD = memtoregE & (rtE == rsD | rtE == rtD);

    assign #1 branchstallD = branchD &

                (regwriteE &

                (writeregE == rsD | writeregE == rtD) |

                memtoregM &

                (writeregM == rsD | writeregM == rtD));

    assign #1 stallE = divstallE;

    assign #1 stallD = lwstallD | branchstallD | stallE;

    assign #1 stallF = stallD;

        //stalling D stalls all previous stages

    //assign #1 flushE = stallD;//???这个啥意思，不太明白

        //stalling D flushes next stage

    // Note: not necessary to stall D stage on store

    //       if source comes from load;

    //       instead, another bypass network could

    //       be added from W to M

endmodule

……

三、实验过程（40%）

（一）设计工作日志

2021.12.21-12.31：复习并参加计算机网络、机器学习、计算机体系结构的考试、完成计算机网络、机器学习、Java 企业级应用的期末报告与课程设计

2022.1.1：阅读相关资料，观看计组实验4的讲解视频以及硬综讲解视频进行学习，并熟悉实验4的代码

2022.1.2：一起完成了逻辑运算指令的实现，并通过了独立测试

2022.1.3：一起完成了算术运算指令（乘除法除外），移位指令，修改和完善controller模块相应的控制信号和相应的数据通路

2022.1.4 王谦铭重新调整冒险，通路和控制信号，完成了乘除法指令和数据移动指令，王渝森熟悉了乘除法指令，并熟悉它的运算过程

2022.1.5 王谦铭完成了跳转指令，并增加相应的控制信号和数据通路

2022.1.6 一起完成了访存指令，并开始所有指令的独立测试，通过调整代码完成了6组独立测试，并尝试封装sram，但未成功。

（二）主要的错误记录

1、错误1

（1）错误现象

执行单独几类指令的功能仿真时，发现部分的数据始终是X；

PC正常了一段时间之后，突然一部分变为了X，导致无法正常取指。

（2）分析定位过程

由于PC变为X可能就是为X的信号造成的，因此先解决了为X的数据/信号，再看PC是否正常变化。

（3）错误原因

根据PPT的内容，有X，要么没初始化；要么多驱动。所以一直为X的原因是信号未定义。

（4）修正效果

添加上信号的定义，并且将连接修改正确后，信号变化正常，PC变化正常。

（5）归纳总结（可选）

这类错误往往是由于信号太多，并且代码写得比较杂乱，因此在写信号时，还需要考虑从整齐直观的角度进行编写。

2、错误2

（1）错误现象

普通分支branch指令跳转正确，但是bal类指令跳转出错。

（2）分析定位过程

bal类的分支指令和普通的branch指令不同之处在于，bal类指令需要写寄存器堆，而普通的branch指令不写寄存器。

所以判断可能是控制分支的冒险模块有错误。根据仿真的波形图，进一步认为是冒险的暂停信号导致的错误。

（3）错误原因

原本冒险模块的分支指令类设计是需要暂停和数据前推的，但是暂停可能导致了出错。

（4）修正效果

重新设计了冒险模块，去掉了分支类的暂停信号，使用旁路来前推数据。修改了冒险模块，并新增加了旁路之后，最终跳转正确。

3、错误3

（1）错误现象

除法运算结果不正确。

（2）分析定位过程

观察仿真波形图发现，除法的一个运算数在运算过程中发生了改变，从而导致了最后的结果发生了错误。

（3）错误原因

由于操作数是通过多选器来选择的，即转发旁路，在除法暂停时，多选器选择的值发生了改变，导致操作数改变了。

（4）修正效果

修改了除法器，将第一次获得的源操作数保存在一个寄存器中，并将这个寄存器的值作为之后运算的操作数，即在除法运算过程中不让操作数发生变化。修改后最终的输出结果正确。

四、设计结果

（一）设计交付物说明

目录层次说明：

外围框架：打包了soc框架

自己的代码：

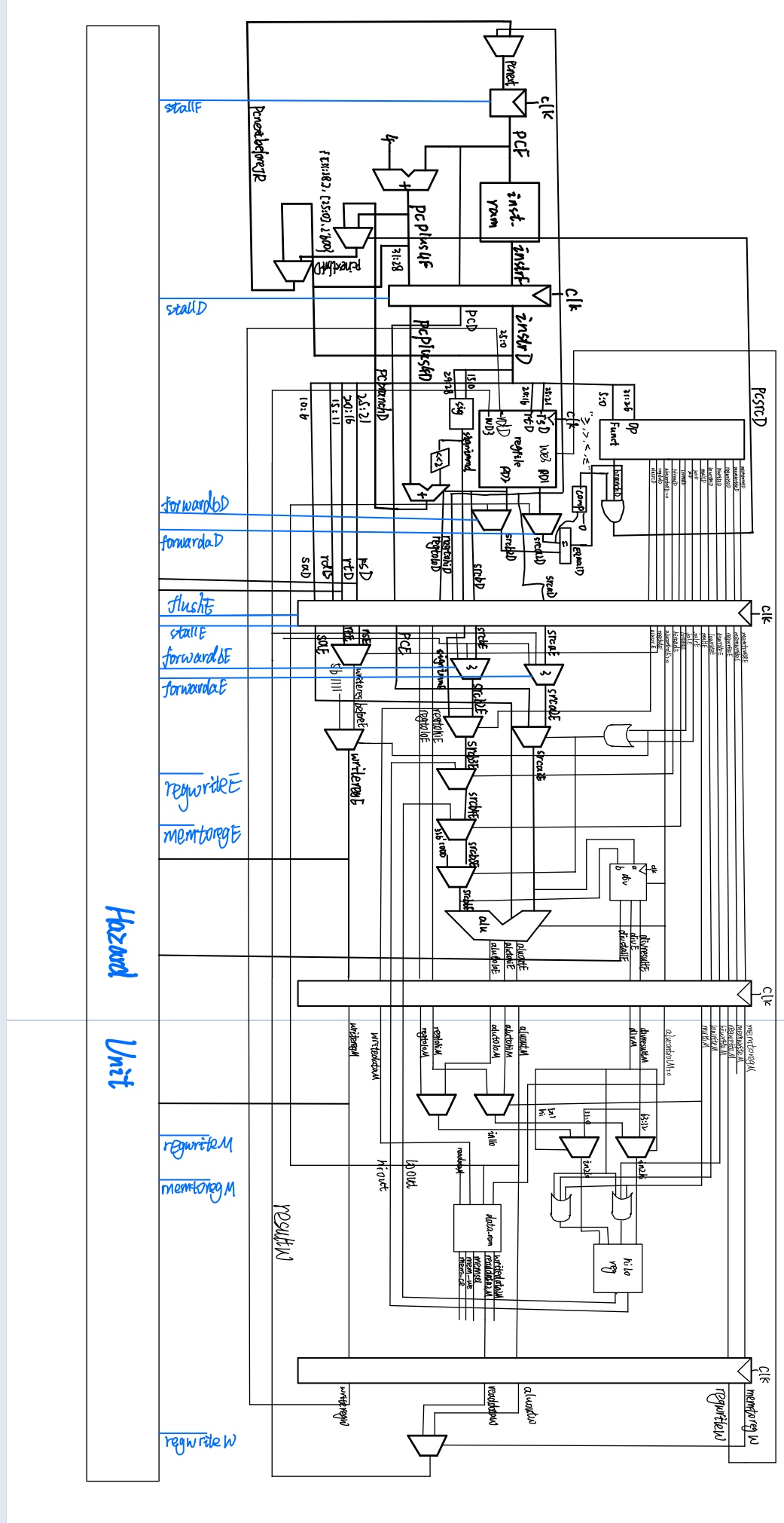
没有修改实验四的代码结构，即使有很多信号量需要来来回回传。指令集和数据寄存器在mips外部，instdec是供仿真看的，mmu是为了调试一个bug加上的，目的是虚拟内存和实际内存的转换。Mips里由controller和datapath组成，两个都是直接拿的实验四的大体结构，controller里，我将maindec拆分成两部分，因为不像实验四只有三位aluop，我们把aluop扩展到了八位，因此为了方便编写和修改代码，把maindec拆成了maindec和aludec

Datapath基本上也复用实验四的框架结构，但是多加了一些组件。首先是多加了一些选择器，这些东西都是因为更多的指令写进了cpu，所以某些接口需要更多的选择。除法器是新加入的一个组件，负责进行除法运算。Hi\_lo寄存器也是新加的组件，负责hilo寄存器的读和写。另外一个组件是changedatamem组件，这个组件主要负责半字，字的load和store，因为读写半字和字，需要对读到的数据进行扩展，需要对写入的数据进行选择，因此我们设计了这样一个组件。



（二）设计演示结果

手绘数据通路图：



五、参考设计说明

数据通路 datapath 模块、controller 模块、hazrad 模块的设计基于实验四的基础上和课程设计提供的原始数据通路图进行扩充。

div.v模块，hilo\_reg.v和宏定义defines.vh模块均参考ref\_code里的代码

六、现场添加指令和答辩记录

（一）现场添加指令

(1) 分工情况

两人共同完成指令的实现

(2) 完成情况

成功完成任务

（二）现场答辩记录

问题1：解释延迟槽：

王渝森：分支指令的后一条指令，不论是否发生分支，都会被执行。

问题2：branch有暂停没？

王谦铭：D和E阶段有数据相同的时候会暂停，暂停后会用流水线刷新，刷新E阶段的数据，从而解除暂停。

问题3：什么阶段处理跳转？

王谦铭：D

问题4：为啥在时钟下降沿写入寄存器

王谦铭：说了一大堆，没说到点子上。

问题5：除法运算？

王谦铭：32个周期，一位一位算，记录每一位得到的除数，算完后可以得到余数

七、总结

（一）组员：王谦铭

挺好玩的。

（二）组员：王渝森

作为一个信安的同学，做这个硬件综合设计还是非常吃力的，而且这学期也没有选体系结构这门课，硬件底子就更差了，而且时间比较紧，等考完试，写完报告已经只剩一周左右了，尽管很难，但在这次硬件综合设计还是收获了很多，对硬件有了更多的了解。

八、参考文献

[1] 雷思磊.自己动手写 CPU.电子工业出版社,2014.

[2] cyyself.lab4 2021.2. https://github.com/cyyself/step\_into\_mips